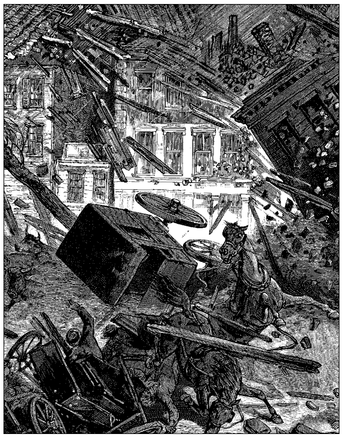
CHAPTER NINE

INTERRUPT HANDLING



Although some devices can be controlled using nothing but their I/O regions, most real-world devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different, and slower, than that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened.

That way, of course, is interrupts. An *interrupt* is simply a signal that the hardware can send when it wants the processor's attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device's interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.

It is difficult to demonstrate the use of interrupts without a real hardware device to generate them. Thus, the sample code used in this chapter works with the parallel port. We'll be working with the *short* module from the previous chapter; with some small additions it can generate and handle interrupts from the parallel port. The module's name, *short*, actually means *short int* (it is C, isn't it?), to remind us that it handles *int*errupts.

Overall Control of Interrupts

The way that Linux handles interrupts has changed quite a bit over the years, due to changes in design and in the hardware it works with. The PC's view of interrupts in the early days was quite simple; there were just 16 interrupt lines and one processor to deal with them. Modern hardware can have many more interrupts, and can also be equipped with fancy advanced programmable interrupt controllers (APICs), which can distribute interrupts across multiple processors in an intelligent (and programmable) way.

Happily, Linux has been able to deal with all of these changes with relatively few incompatibilities at the driver level. Thus, the interface described in this chapter works, with few differences, across many kernel versions. Sometimes things do work out nicely.

Unix-like systems have used the functions *cli* and *sti* to disable and enable interrupts for many years. In modern Linux systems, however, using them directly is discouraged. It is increasingly impossible for any routine to know whether interrupts are enabled when it is called; thus, simply enabling interrupts with *sti* before return is a bad practice. Your function may be returning to a function that expects interrupts to be still disabled.

Thus, if you must disable interrupts, it is better to use the following calls:

```
unsigned long flags;

    save_flags(flags);
    cli();

    /* This code runs with interrupts disabled */

    restore_flags(flags);
```

Note that *save_flags* is a macro, and that it is passed the variable to hold the flags directly—without an `&` operator. There is also an important constraint on the use of these macros: *save_flags* and *restore_flags* must be called from the same function. In other words, you cannot pass the `flags` to another function, unless the other function is inlined. Code that ignores this restriction will work on some architectures but will fail on others.

Increasingly, however, even code like the previous example is discouraged wherever possible. In a multiprocessor system, critical code cannot be protected just by disabling interrupts; some sort of locking mechanism must be used. Functions such as *spin_lock_irqsave* (covered in “Using Spinlocks,” later in this chapter) provide locking and interrupt control together; these functions are the only really safe way to control concurrency in the presence of interrupts.

cli, meanwhile, disables interrupts on *all* processors on the system, and can thus affect the performance of the system as a whole.*

---

* The truth is just a little more complicated than this. If you are already handling an interrupt, *cli* only disables interrupts on the current CPU.

Thus, explicit calls to *cli* and related functions are slowly disappearing from much of the kernel. There are occasions where you need them in a device driver, but they are rare. Before calling *cli*, think about whether you *really* need to disable all interrupts on the system.

Preparing the Parallel Port

Although the parallel interface is simple, it can trigger interrupts. This capability is used by the printer to notify the *lp* driver that it is ready to accept the next character in the buffer.

Like most devices, the parallel port doesn't actually generate interrupts before it's instructed to do so; the parallel standard states that setting bit 4 of port 2 (`0x37a`, `0x27a`, or whatever) enables interrupt reporting. A simple *outb* call to set the bit is performed by *short* at module initialization.

Once interrupts are enabled, the parallel interface generates an interrupt whenever the electrical signal at pin 10 (the so-called ACK bit) changes from low to high. The simplest way to force the interface to generate interrupts (short of hooking up a printer to the port) is to connect pins 9 and 10 of the parallel connector. A short length of wire inserted into the appropriate holes in the parallel port connector on the back of your system will create this connection. The pinout of the parallel port is shown in Figure 8-1.

Pin 9 is the most significant bit of the parallel data byte. If you write binary data to */dev/short0*, you'll generate several interrupts. Writing ASCII text to the port won't generate interrupts, though, because the most significant bit won't be set.

If you'd rather avoid soldering, but you do have a printer at hand, you can run the sample interrupt handler using a real printer, as shown later. Note, however, that the probing functions we are going to introduce depend on the jumper between pin 9 and 10 being in place, and you'll need it to experiment with probing using our code.

Installing an Interrupt Handler

If you want to actually "see" interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect your interrupt, it will simply acknowledge and ignore it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it, and to release it when it's done. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in `<linux/sched.h>`, implement the interface:

```
int request_irq(unsigned int irq,
   void (*handler)(int, void *, struct pt_regs *),
   unsigned long flags,
   const char *dev_name,
   void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

The value returned from *request_irq* to the requesting function is either 0 to indicate success or a negative error code, as usual. It's not uncommon for the function to return `-EBUSY` to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

`unsigned int irq`
: This is the interrupt number being requested.

`void (*handler)(int, void *, struct pt_regs *)`
: The pointer to the handling function being installed. We'll discuss the arguments to this function later in this chapter.

`unsigned long flags`
: As you might expect, a bit mask of options (described later) related to interrupt management.

`const char *dev_name`
: The string passed to *request_irq* is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

`void *dev_id`
: This pointer is used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). When no sharing is in force, `dev_id` can be set to `NULL`, but it a good idea anyway to use this item to point to the device structure. We'll see a practical use for `dev_id` in "Implementing a Handler," later in this chapter.

The bits that can be set in `flags` are as follows:

`SA_INTERRUPT`
: When set, this indicates a "fast" interrupt handler. Fast handlers are executed with interrupts disabled (the topic is covered in deeper detail later in this chapter, in "Fast and Slow Handlers").

`SA_SHIRQ`

This bit signals that the interrupt can be shared between devices. The concept of sharing is outlined in "Interrupt Sharing," later in this chapter.

`SA_SAMPLE_RANDOM`

This bit indicates that the generated interrupts can contribute to the entropy pool used by */dev/random* and */dev/urandom*. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption. Such random numbers are extracted from an entropy pool that is contributed by various random events. If your device generates interrupts at truly random times, you should set this flag. If, on the other hand, your interrupts will be predictable (for example, vertical blanking of a frame grabber), the flag is not worth setting—it wouldn't contribute to system entropy anyway. Devices that could be influenced by attackers should not set this flag; for example, network drivers can be subjected to predictable packet timing from outside and should not contribute to the entropy pool. See the comments in *drivers/char/random.c* for more information.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea, it actually isn't. Because the number of interrupt lines is limited, you don't want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used. Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

It is possible, for example, to run a frame grabber on the same interrupt as a modem, as long as you don't use the two devices at the same time. It is quite common for users to load the module for a special device at system boot, even if the device is rarely used. A data acquisition gadget might use the same interrupt as the second serial port. While it's not too hard to avoid connecting to your Internet service provider (ISP) during data acquisition, being forced to unload a module in order to use the modem is really unpleasant.

The correct place to call *request_irq* is when the device is first opened, *before* the hardware is instructed to generate interrupts. The place to call *free_irq* is the last time the device is closed, *after* the hardware is told not to interrupt the processor any more. The disadvantage of this technique is that you need to keep a per-device open count. Using the module count isn't enough if you control two or more devices from the same module.

This discussion notwithstanding, *short* requests its interrupt line at load time. This was done so that you can run the test programs without having to run an extra process to keep the device open. *short*, therefore, requests the interrupt from within its initialization function (*short_init*) instead of doing it in *short_open*, as a real device driver would.

The interrupt requested by the following code is `short_irq`. The actual assignment of the variable (i.e., determining which IRQ to use) is shown later, since it is not relevant to the current discussion. `short_base` is the base I/O address of the parallel interface being used; register 2 of the interface is written to enable interrupt reporting.

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
                         SA_INTERRUPT, "short", NULL);
    if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
               short_irq);
        short_irq = -1;
    }
    else { /* actually enable it -- assume this *is* a parallel port */
        outb(0x10,short_base+2);
    }
}
```

The code shows that the handler being installed is a fast handler (`SA_INTERRUPT`), does not support interrupt sharing (`SA_SHIRQ` is missing), and doesn't contribute to system entropy (`SA_SAMPLE_RANDOM` is missing too). The *outb* call then enables interrupt reporting for the parallel port.

The /proc Interface

Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected. Reported interrupts are shown in */proc/interrupts*. The following snapshot was taken after several days of uptime on a two-processor Pentium system:

```
           CPU0       CPU1
  0:   34584323   34936135    IO-APIC-edge  timer
  1:     224407     226473    IO-APIC-edge  keyboard
  2:          0          0          XT-PIC  cascade
  5:    5636751    5636666   IO-APIC-level  eth0
  9:          0          0   IO-APIC-level  acpi
 10:     565910     565269   IO-APIC-level  aic7xxx
 12:     889091     884276    IO-APIC-edge  PS/2 Mouse
 13:          1          0          XT-PIC  fpu
 15:    1759669    1734520    IO-APIC-edge  ide1
NMI:   69520392   69520392
LOC:   69513717   69513716
ERR:          0
```

The first column is the IRQ number. You can see from the IRQs that are missing that the file shows only interrupts corresponding to installed handlers. For example, the first serial port (which uses interrupt number 4) is not shown, indicating that the modem isn't being used. In fact, even if the modem had been used earlier but wasn't in use at the time of the snapshot, it would not show up in the file; the serial ports are well behaved and release their interrupt handlers when the device is closed.

The */proc/interrupts* display shows how many interrupts have been delivered to each CPU on the system. As you can see from the output, the Linux kernel tries to divide interrupt traffic evenly across the processors, with some success. The final columns give information on the programmable interrupt controller that handles the interrupt (and which a driver writer need not worry about), and the name(s) of the device(s) that have registered handlers for the interrupt (as specified in the `dev_name` argument to *request_irq*).

The */proc* tree contains another interrupt-related file, */proc/stat*; sometimes you'll find one file more useful and sometimes you'll prefer the other. */proc/stat* records several low-level statistics about system activity, including (but not limited to) the number of interrupts received since system boot. Each line of *stat* begins with a text string that is the key to the line; the `intr` mark is what we are looking for. The following (truncated and line-broken) snapshot was taken shortly after the previous one:

```
intr 884865 695557 4527 0 3109 4907 112759 3 0 0 0 11314
     0 17747 1 0 34941 0 0 0 0 0 0 0
```

The first number is the total of all interrupts, while each of the others represents a single IRQ line, starting with interrupt 0. This snapshot shows that interrupt number 4 has been used 4907 times, even though no handler is *currently* installed. If the driver you're testing acquires and releases the interrupt at each open and close cycle, you may find */proc/stat* more useful than */proc/interrupts*.

Another difference between the two files is that *interrupts* is not architecture dependent, whereas *stat* is: the number of fields depends on the hardware underlying the kernel. The number of available interrupts varies from as few as 15 on the SPARC to as many as 256 on the IA-64 and a few other systems. It's interesting to note that the number of interrupts defined on the x86 is currently 224, not 16 as you may expect; this, as explained in *include/asm-i386/irq.h*, depends on Linux using the architectural limit instead of an implementation-specific limit (like the 16 interrupt sources of the old-fashioned PC interrupt controller).

The following is a snapshot of */proc/interrupts* taken on an IA-64 system. As you can see, besides different hardware routing of common interrupt sources, there's no platform dependency here.

```
           CPU0       CPU1
 27:       1705      34141  IO-SAPIC-level  qla1280
 40:          0          0           SAPIC  perfmon
 43:        913       6960  IO-SAPIC-level  eth0
 47:      26722        146  IO-SAPIC-level  usb-uhci
 64:          3          6   IO-SAPIC-edge  ide0
 80:          4          2   IO-SAPIC-edge  keyboard
 89:          0          0   IO-SAPIC-edge  PS/2 Mouse
239:    5606341    5606052           SAPIC  timer
254:      67575      52815           SAPIC  IPI
NMI:          0          0
ERR:          0
```

Autodetecting the IRQ Number

One of the most compelling problems for a driver at initialization time can be how to determine which IRQ line is going to be used by the device. The driver needs the information in order to correctly install the handler. Even though a programmer could require the user to specify the interrupt number at load time, this is a bad practice because most of the time the user doesn't know the number, either because he didn't configure the jumpers or because the device is jumperless. Autodetection of the interrupt number is a basic requirement for driver usability.

Sometimes autodetection depends on the knowledge that some devices feature a default behavior that rarely, if ever, changes. In this case, the driver might assume that the default values apply. This is exactly how *short* behaves by default with the parallel port. The implementation is straightforward, as shown by *short* itself:

```
if (short_irq < 0) /* not yet specified: force the default on */
    switch(short_base) {
      case 0x378: short_irq = 7; break;
      case 0x278: short_irq = 2; break;
      case 0x3bc: short_irq = 5; break;
    }
```

The code assigns the interrupt number according to the chosen base I/O address, while allowing the user to override the default at load time with something like

```
insmod ./short.o short_irq=x.
```

`short_base` defaults to `0x378`, so `short_irq` defaults to 7.

Some devices are more advanced in design and simply "announce" which interrupt they're going to use. In this case, the driver retrieves the interrupt number by reading a status byte from one of the device's I/O ports or PCI configuration space. When the target device is one that has the ability to tell the driver which interrupt it is going to use, autodetecting the IRQ number just means probing the device, with no additional work required to probe the interrupt.

It's interesting to note here that modern devices supply their interrupt configuration. The PCI standard solves the problem by requiring peripheral devices to declare what interrupt line(s) they are going to use. The PCI standard is discussed in Chapter 15.

Unfortunately, not every device is programmer friendly, and autodetection might require some probing. The technique is quite simple: the driver tells the device to generate interrupts and watches what happens. If everything goes well, only one interrupt line is activated.

Though probing is simple in theory, the actual implementation might be unclear. We'll look at two ways to perform the task: calling kernel-defined helper functions and implementing our own version.

Kernel-assisted probing

The Linux kernel offers a low-level facility for probing the interrupt number. It only works for nonshared interrupts, but then most hardware that is capable of working in a shared interrupt mode provides better ways of finding the configured interrupt number. The facility consists of two functions, declared in `<linux/interrupt.h>` (which also describes the probing machinery):

`unsigned long probe_irq_on(void);`
: This function returns a bit mask of unassigned interrupts. The driver must preserve the returned bit mask and pass it to *probe_irq_off* later. After this call, the driver should arrange for its device to generate at least one interrupt.

`int probe_irq_off(unsigned long);`
: After the device has requested an interrupt, the driver calls this function, passing as argument the bit mask previously returned by *probe_irq_on*. *probe_irq_off* returns the number of the interrupt that was issued after "probe_on." If no interrupts occurred, 0 is returned (thus, IRQ 0 can't be probed for, but no custom device can use it on any of the supported architectures anyway). If more than one interrupt occurred (ambiguous detection), *probe_irq_off* returns a negative value.

The programmer should be careful to enable interrupts on the device *after* the call to *probe_irq_on* and to disable them *before* calling *probe_irq_off*, Additionally, you must remember to service the pending interrupt in your device after *probe_irq_off*.

The *short* module demonstrates how to use such probing. If you load the module with `probe=1`, the following code is executed to detect your interrupt line, provided pins 9 and 10 of the parallel connector are bound together:

```
int count = 0;
do {
    unsigned long mask;

    mask = probe_irq_on();
    outb_p(0x10,short_base+2); /* enable reporting */
    outb_p(0x00,short_base);   /* clear the bit */
    outb_p(0xFF,short_base);   /* set the bit: interrupt! */
    outb_p(0x00,short_base+2); /* disable reporting */
```

```
        udelay(5);  /* give it some time */
        short_irq = probe_irq_off(mask);

        if (short_irq == 0) { /* none of them? */
            printk(KERN_INFO "short: no irq reported by probe\n");
            short_irq = -1;
        }
        /*
         * If more than one line has been activated, the result is
         * negative. We should service the interrupt (no need for lpt port)
         * and loop over again. Loop at most five times, then give up
         */
    } while (short_irq < 0 && count++ < 5);
    if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

Note the use of *udelay* before calling *probe_irq_off*. Depending on the speed of your processor, you may have to wait for a brief period to give the interrupt time to actually be delivered.

If you dig through the kernel sources, you may stumble across references to a different pair of functions:

`void autoirq_setup(int waittime);`
Set up for an IRQ probe. The `waittime` argument is not used.

`int autoirq_report(int waittime);`
Delays for the given interval (in jiffies), then returns the number of the IRQ seen since *autoirq_setup* was called.

These functions are used primarily in the network driver code, for historical reasons. They are currently implemented with *probe_irq_on* and *probe_irq_off*; there is not usually any reason to use the *autoirq_* functions over the *probe_irq_* functions.

Probing might be a lengthy task. While this is not true for *short*, probing a frame grabber, for example, requires a delay of at least 20 ms (which is ages for the processor), and other devices might take even longer. Therefore, it's best to probe for the interrupt line only once, at module initialization, independently of whether you install the handler at device open (as you should) or within the initialization function (which is not recommended).

It's interesting to note that on some platforms (PowerPC, M68k, most MIPS implementations, and both SPARC versions), probing is unnecessary and therefore the previous functions are just empty placeholders, sometimes called "useless ISA nonsense." On other platforms, probing is only implemented for ISA devices. Anyway, most architectures define the functions (even if empty) to ease porting existing device drivers.

Generally speaking, probing is a hack, and mature architectures are like the PCI bus, which provides all the needed information.

Do-it-yourself probing

Probing can be implemented in the driver itself without too much trouble. The *short* module performs do-it-yourself detection of the IRQ line if it is loaded with `probe=2`.

The mechanism is the same as the one described earlier: enable all unused interrupts, then wait and see what happens. We can, however, exploit our knowledge of the device. Often a device can be configured to use one IRQ number from a set of three or four; probing just those IRQs enables us to detect the right one, without having to test for all possible IRQs.

The *short* implementation assumes that 3, 5, 7, and 9 are the only possible IRQ values. These numbers are actually the values that some parallel devices allow you to select.

The following code probes by testing all "possible" interrupts and looking at what happens. The `trials` array lists the IRQs to try and has `0` as the end marker; the `tried` array is used to keep track of which handlers have actually been registered by this driver.

```
int trials[] = {3, 5, 7, 9, 0};
int tried[]  = {0, 0, 0, 0, 0};
int i, count = 0;

 /*
  * Install the probing handler for all possible lines. Remember
  * the result (0 for success, or -EBUSY) in order to only free
  * what has been acquired
  */
for (i=0; trials[i]; i++)
    tried[i] = request_irq(trials[i], short_probing,
                           SA_INTERRUPT, "short probe", NULL);

do {
    short_irq = 0; /* none obtained yet */
    outb_p(0x10,short_base+2); /* enable */
    outb_p(0x00,short_base);
    outb_p(0xFF,short_base); /* toggle the bit */
    outb_p(0x00,short_base+2); /* disable */
    udelay(5);  /* give it some time */

    /* the value has been set by the handler */
    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
     * If more than one line has been activated, the result is
     * negative. We should service the interrupt (but the lpt port
     * doesn't need it) and loop over again. Do it at most 5 times
     */
} while (short_irq <=0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i=0; trials[i]; i++)
    if (tried[i] == 0)
        free_irq(trials[i], NULL);

if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

You might not know in advance what the "possible" IRQ values are. In that case, you'll need to probe all the free interrupts, instead of limiting yourself to a few `trials[]`. To probe for all interrupts, you have to probe from IRQ 0 to IRQ `NR_IRQS-1`, where `NR_IRQS` is defined in `<asm/irq.h>` and is platform dependent.

Now we are missing only the probing handler itself. The handler's role is to update `short_irq` according to which interrupts are actually received. A 0 value in `short_irq` means "nothing yet," while a negative value means "ambiguous." These values were chosen to be consistent with *probe_irq_off* and to allow the same code to call either kind of probing within *short.c*.

```
void short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq == 0) short_irq = irq;    /* found */
    if (short_irq != irq) short_irq = -irq; /* ambiguous */
}
```

The arguments to the handler are described later. Knowing that `irq` is the interrupt being handled should be sufficient to understand the function just shown.

Fast and Slow Handlers

Older versions of the Linux kernel took great pains to distinguish between "fast" and "slow" interrupts. Fast interrupts were those that could be handled very quickly, whereas handling slow interrupts took significantly longer. Slow interrupts could be sufficiently demanding of the processor that it was worthwhile to reenable interrupts while they were being handled. Otherwise, tasks requiring quick attention could be delayed for too long.

In modern kernels most of the differences between fast and slow interrupts have disappeared. There remains only one: fast interrupts (those that were requested with the `SA_INTERRUPT` flag) are executed with all other interrupts disabled on the current processor. Note that other processors can still handle interrupts, though you will never see two processors handling the same IRQ at the same time.

To summarize the slow and fast executing environments:

- A fast handler runs with interrupt reporting disabled in the microprocessor, and the interrupt being serviced is disabled in the interrupt controller. The handler can nonetheless enable reporting in the processor by calling *sti*.
- A slow handler runs with interrupt reporting enabled in the processor, and the interrupt being serviced is disabled in the interrupt controller.

So, which type of interrupt should your driver use? On modern systems, `SA_INTERRUPT` is only intended for use in a few, specific situations (such as timer interrupts). Unless you have a strong reason to run your interrupt handler with other interrupts disabled, you should not use `SA_INTERRUPT`.

This description should satisfy most readers, though someone with a taste for hardware and some experience with her computer might be interested in going deeper. If you don't care about the internal details, you can skip to the next section.

The internals of interrupt handling on the x86

This description has been extrapolated from *arch/i386/kernel/irq.c*, *arch/i386/kernel/i8259.c*, and *include/asm-i386/hw_irq.h* as they appear in the 2.4 kernels; although the general concepts remain the same, the hardware details differ on other platforms.

The lowest level of interrupt handling resides in assembly code declared as macros in *hw_irq.h* and expanded in *i8259.c*. Each interrupt is connected to the function *do_IRQ*, defined in *irq.c*.

The first thing *do_IRQ* does is to acknowledge the interrupt so that the interrupt controller can go on to other things. It then obtains a spinlock for the given IRQ number, thus preventing any other CPU from handling this IRQ. It clears a couple of status bits (including one called `IRQ_WAITING` that we'll look at shortly), and then looks up the handler(s) for this particular IRQ. If there is no handler, there's nothing to do; the spinlock is released, any pending tasklets and bottom halves are run, and *do_IRQ* returns.

Usually, however, if a device is interrupting there is a handler registered as well. The function *handle_IRQ_event* is called to actually invoke the handlers. It starts by testing a global interrupt lock bit; if that bit is set, the processor will spin until it is cleared. Calling *cli* sets this bit, thus blocking handling of interrupts; the normal interrupt handling mechanism does *not* set this bit, and thus allows further processing of interrupts. If the handler is of the slow variety, interrupts are reenabled in the hardware and the handler is invoked. Then it's just a matter of cleaning up, running tasklets and bottom halves, and getting back to regular work. The "regular work" may well have changed as a result of an interrupt (the handler could *wake_up* a process, for example), so the last thing that happens on return from an interrupt is a possible rescheduling of the processor.

Probing for IRQs is done by setting the `IRQ_WAITING` status bit for each IRQ that currently lacks a handler. When the interrupt happens, *do_IRQ* clears that bit and then returns, since no handler is registered. *probe_irq_off*, when called by a driver, need only search for the IRQ that no longer has `IRQ_WAITING` set.

Implementing a Handler

So far, we've learned to register an interrupt handler, but not to write one. Actually, there's nothing unusual about a handler—it's ordinary C code.

The only peculiarity is that a handler runs at interrupt time and therefore suffers some restrictions on what it can do. These restrictions are the same as those we saw with task queues. A handler can't transfer data to or from user space, because it doesn't execute in the context of a process. Handlers also cannot do anything that would sleep, such as calling *sleep_on*, allocating memory with anything other than `GFP_ATOMIC`, or locking a semaphore. Finally, handlers cannot call *schedule*.

The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board; most hardware devices won't generate other interrupts until their "interrupt-pending" bit has been cleared. Some devices don't require this step because they don't have an "interrupt-pending" bit; such devices are a minority, although the parallel port is one of them. For that reason, *short* does not have to clear such a bit.

A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for, such as the arrival of new data.

To stick with the frame grabber example, a process could acquire a sequence of images by continuously reading the device; the *read* call blocks before reading each frame, while the interrupt handler awakens the process as soon as each new frame arrives. This assumes that the grabber interrupts the processor to signal successful arrival of each new frame.

The programmer should be careful to write a routine that executes in a minimum of time, independent of its being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a tasklet or task queue to schedule computation at a safer time (see "Task Queues" in Chapter 6).

Our sample code in *short* makes use of the interrupt to call *do_gettimeofday* and print the current time to a page-sized circular buffer. It then awakens any reading process because there is now data available to be read.

```
void short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    int written;

    do_gettimeofday(&tv);

    /* Write a 16-byte record. Assume PAGE_SIZE is a multiple of 16 */
    written = sprintf((char *)short_head,"%08u.%06u\n",
                  (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* wake any reading process */
}
```

This code, though simple, represents the typical job of an interrupt handler. It, in turn, calls *short_incr_bp*, which is defined as follows:

```
static inline void short_incr_bp(volatile unsigned long *index,
                                 int delta)
{
    unsigned long new = *index + delta;
    barrier ();  /* Don't optimize these two together */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

This function has been carefully written to wrap a pointer into the circular buffer without ever exposing an incorrect value. By assigning only the final value and placing a barrier to keep the compiler from optimizing things, it is possible to manipulate the circular buffer pointers safely without locks.

The device file used to read the buffer being filled at interrupt time is */dev/shortint*. This device special file, together with */dev/shortprint*, wasn't introduced in Chapter 8, because its use is specific to interrupt handling. The internals of */dev/shortint* are specifically tailored for interrupt generation and reporting. Writing to the device generates one interrupt every other byte; reading the device gives the time when each interrupt was reported.

If you connect together pins 9 and 10 of the parallel connector, you can generate interrupts by raising the high bit of the parallel data byte. This can be accomplished by writing binary data to */dev/short0* or by writing anything to */dev/shortint*.*

The following code implements *read* and *write* for */dev/shortint*.

---

* The *shortint* device accomplishes its task by alternately writing 0x00 and 0xff to the parallel port.

```
ssize_t short_i_read (struct file *filp, char *buf, size_t count,
                      loff_t *f_pos)
{
    int count0;

    while (short_head == short_tail) {
        interruptible_sleep_on(&short_queue);
        if (signal_pending (current))  /* a signal arrived */
          return -ERESTARTSYS; /* tell the fs layer to handle it */
        /* else, loop */
    }
    /* count0 is the number of readable data bytes */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))
       return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;
}

ssize_t short_i_write (struct file *filp, const char *buf, size_t count,
                loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long address = short_base; /* output to the parallel
                                           data latch */

    if (use_mem) {
       while (written < count)
           writeb(0xff * ((++written + odd) & 1), address);
    } else {
       while (written < count)
           outb(0xff * ((++written + odd) & 1), address);
    }

    *f_pos += count;
    return written;
}
```

The other device special file, */dev/shortprint*, uses the parallel port to drive a printer, and you can use it if you want to avoid soldering a wire between pin 9 and 10 of a D-25 connector. The *write* implementation of *shortprint* uses a circular buffer to store data to be printed, while the *read* implementation is the one just shown (so you can read the time your printer takes to eat each character).

In order to support printer operation, the interrupt handler has been slightly modified from the one just shown, adding the ability to send the next data byte to the printer if there is more data to transfer.

Using Arguments

Though *short* ignores them, three arguments are passed to an interrupt handler: `irq`, `dev_id`, and `regs`. Let's look at the role of each.

The interrupt number (`int irq`) is useful as information you may print in your log messages, if any. Although it had a role in pre-2.0 kernels, when no `dev_id` existed, `dev_id` serves that role much better.

The second argument, `void *dev_id`, is a sort of ClientData; a `void *` argument is passed to *request_irq*, and this same pointer is then passed back as an argument to the handler when the interrupt happens.

You'll usually pass a pointer to your device data structure in `dev_id`, so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event. Typical use of the argument in an interrupt handler is as follows:

```
static void sample_interrupt(int irq, void *dev_id, struct pt_regs
                             *regs)
{
    struct sample_dev *dev = dev_id;

    /* now ‘dev’ points to the right hardware item */
    /* .... */
}
```

The typical *open* code associated with this handler looks like this:

```
static void sample_open(struct inode *inode, struct file *filp)
{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt,
    0 /* flags */, "sample", dev /* dev_id */);
    /*....*/
    return 0;
}
```

The last argument, `struct pt_regs *regs`, is rarely used. It holds a snapshot of the processor's context before the processor entered interrupt code. The registers can be used for monitoring and debugging; they are not normally needed for regular device driver tasks.

Enabling and Disabling Interrupts

We have already seen the *sti* and *cli* functions, which can enable and disable all interrupts. Sometimes, however, it's useful for a driver to enable and disable interrupt reporting for its own IRQ line only. The kernel offers three functions for this purpose, all declared in `<asm/irq.h>`:

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

Calling any of these functions may update the mask for the specified `irq` in the programmable interrupt controller (PIC), thus disabling or enabling IRQs across all processors. Calls to these functions can be nested—if *disable_irq* is called twice in succession, two *enable_irq* calls will be required before the IRQ is truly reenabled. It is possible to call these functions from an interrupt handler, but enabling your own IRQ while handling it is not usually good practice.

disable_irq will not only disable the given interrupt, but will also wait for a currently executing interrupt handler, if any, to complete. *disable_irq_nosync*, on the other hand, returns immediately. Thus, using the latter will be a little faster, but may leave your driver open to race conditions.

But why disable an interrupt? Sticking to the parallel port, let's look at the *plip* network interface. A *plip* device uses the bare-bones parallel port to transfer data. Since only five bits can be read from the parallel connector, they are interpreted as four data bits and a clock/handshake signal. When the first four bits of a packet are transmitted by the initiator (the interface sending the packet), the clock line is raised, causing the receiving interface to interrupt the processor. The *plip* handler is then invoked to deal with newly arrived data.

After the device has been alerted, the data transfer proceeds, using the handshake line to clock new data to the receiving interface (this might not be the best implementation, but it is necessary for compatibility with other packet drivers using the parallel port). Performance would be unbearable if the receiving interface had to handle two interrupts for every byte received. The driver therefore disables the interrupt during the reception of the packet; instead, a poll-and-delay loop is used to bring in the data.

Similarly, since the handshake line from the receiver to the transmitter is used to acknowledge data reception, the transmitting interface disables its IRQ line during packet transmission.

Finally, it's interesting to note that the SPARC and M68k implementations define both the *disable_irq* and *enable_irq* symbols as pointers rather than functions. This trick allows the kernel to assign the pointers at boot time according to the actual platform being run. The C-language semantics to use the function are the same on all Linux systems, independent of whether this trick is used or not, which helps avoid some tedious coding of conditionals.

Tasklets and Bottom-Half Processing

One of the main problems with interrupt handling is how to perform longish tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt—the one you register with *request_irq*. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time. The use of the term bottom half in the 2.4 kernel can be a bit confusing, in that it can mean either the second half of an interrupt handler or one of the mechanisms used to implement this second half, or both. When we refer to a *bottom half* we are speaking generally about a bottom half; the old Linux bottom-half implementation is referred to explicitly with the acronym BH.

But what is a bottom half useful for?

The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

One thing to keep in mind with bottom-half processing is that all of the restrictions that apply to interrupt handlers also apply to bottom halves. Thus, bottom halves cannot sleep, cannot access user space, and cannot invoke the scheduler.

The Linux kernel has two different mechanisms that may be used to implement bottom-half processing. Tasklets were introduced late in the 2.3 development series; they are now the preferred way to do bottom-half processing, but they are not portable to earlier kernel versions. The older bottom-half (BH) implementation exists in even very old kernels, though it is implemented with tasklets in 2.4. We'll look at both mechanisms here. In general, device drivers writing new code should choose tasklets for their bottom-half processing if possible, though portability considerations may determine that the BH mechanism needs to be used instead.

The following discussion works, once again, with the *short* driver. When loaded with a module option, *short* can be told to do interrupt processing in a top/bottom-half mode, with either a tasklet or bottom-half handler. In this case, the top half executes quickly; it simply remembers the current time and schedules the bottom half processing. The bottom half is then charged with encoding this time and awakening any user processes that may be waiting for data.

Tasklets

We have already had an introduction to tasklets in Chapter 6, so a quick review should suffice here. Remember that tasklets are a special function that may be scheduled to run, in interrupt context, at a system-determined safe time. They may be scheduled to run multiple times, but will only run once. No tasklet will ever run in parallel with itself, since they only run once, but tasklets can run in parallel with other tasklets on SMP systems. Thus, if your driver has multiple tasklets, they must employ some sort of locking to avoid conflicting with each other.

Tasklets are also guaranteed to run on the same CPU as the function that first schedules them. An interrupt handler can thus be secure that a tasklet will not begin executing before the handler has completed. However, another interrupt can certainly be delivered while the tasklet is running, so locking between the tasklet and the interrupt handler may still be required.

Tasklets must be declared with the `DECLARE_TASKLET` macro:

```
DECLARE_TASKLET(name, function, data);
```

`name` is the name to be given to the tasklet, `function` is the function that is called to execute the tasklet (it takes one `unsigned long` argument and returns `void`), and `data` is an unsigned long value to be passed to the tasklet function.

The *short* driver declares its tasklet as follows:

```
void short_do_tasklet (unsigned long);
DECLARE_TASKLET (short_tasklet, short_do_tasklet, 0);
```

The function *tasklet_schedule* is used to schedule a tasklet for running. If *short* is loaded with `tasklet=1`, it installs a different interrupt handler that saves data and schedules the tasklet as follows:

```
void short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* cast to stop
    'volatile' warning */
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_bh_count++; /* record that an interrupt arrived */
}
```

The actual tasklet routine, *short_do_tasklet*, will be executed shortly at the system's convenience. As mentioned earlier, this routine performs the bulk of the work of handling the interrupt; it looks like this:

```
void short_do_tasklet (unsigned long unused)
{
    int savecount = short_bh_count, written;
    short_bh_count = 0; /* we have already been removed from queue */
    /*
     * The bottom half reads the tv array, filled by the top half,
     * and prints it to the circular text buffer, which is then consumed
     * by reading processes
     */

    /* First write the number of interrupts that occurred before
       this bh */

    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
    short_incr_bp(&short_head, written);

    /*
     * Then, write the time values. Write exactly 16 bytes at a time,
     * so it aligns with PAGE_SIZE
     */

    do {
        written = sprintf((char *)short_head,"%08u.%06u\n",
                        (int)(tv_tail->tv_sec % 100000000),
                        (int)(tv_tail->tv_usec));
        short_incr_bp(&short_head, written);
        short_incr_tv(&tv_tail);
    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /* wake any reading process */
}
```

Among other things, this tasklet makes a note of how many interrupts have arrived since it was last called. A device like *short* can generate a great many interrupts in a brief period, so it is not uncommon for several to arrive before the bottom half is executed. Drivers must always be prepared for this possibility, and must be able to determine how much work there is to perform from the information left by the top half.

The BH Mechanism

Unlike tasklets, old-style BH bottom halves have been around almost as long as the Linux kernel itself. They show their age in a number of ways. For example, all BH bottom halves are predefined in the kernel, and there can be a maximum of 32 of them. Since they are predefined, bottom halves cannot be used directly by modules, but that is not actually a problem, as we will see.

Whenever some code wants to schedule a bottom half for running, it calls *mark_bh*. In the older BH implemention, *mark_bh* would set a bit in a bit mask, allowing the corresponding bottom-half handler to be found quickly at runtime. In modern kernels, it just calls *tasklet_schedule* to schedule the bottom-half routine for execution.

Marking bottom halves is defined in `<linux/interrupt.h>` as

```
void mark_bh(int nr);
```

Here, `nr` is the "number" of the BH to activate. The number is a symbolic constant defined in `<linux/interrupt.h>` that identifies the bottom half to run. The function that corresponds to each bottom half is provided by the driver that owns the bottom half. For example, when `mark_bh(SCSI_BH)` is called, the function being scheduled for execution is *scsi_bottom_half_handler*, which is part of the SCSI driver.

As mentioned earlier, bottom halves are static objects, so a modularized driver won't be able to register its *own* BH. There's no support for dynamic allocation of BH bottom halves, and it's unlikely there ever will be. Fortunately, the immediate task queue can be used instead.

The rest of this section lists some of the most interesting bottom halves. It then describes how the kernel runs a BH bottom half, which you should understand in order to use bottom halves properly.

Several BH bottom halves declared by the kernel are interesting to look at, and a few can even be used by a driver, as introduced earlier. These are the most interesting BHs:

`IMMEDIATE_BH`

This is the most important bottom half for driver writers. The function being scheduled runs (with *run_task_queue*) the `tq_immediate` task queue. A driver (like a custom module) that doesn't own a bottom half can use the immediate queue as if it were its own BH. After registering a task in the queue, the driver must mark the BH in order to have its code actually executed; how to do this was introduced in "The immediate queue," in Chapter 6.

`TQUEUE_BH`

This BH is activated at each timer tick *if* a task is registered in `tq_timer`. In practice, a driver can implement its own BH using `tq_timer`. The timer queue introduced in "The timer queue" in Chapter 6 is a BH, but there's no need to call *mark_bh* for it.

`TIMER_BH`

This BH is marked by *do_timer*, the function in charge of the clock tick. The function that this BH executes is the one that drives the kernel timers. There is no way to use this facility for a driver short of using *add_timer*.

The remaining BH bottom halves are used by specific kernel drivers. There are no entry points in them for a module, and it wouldn't make sense for there to be any. The list of these other bottom halves is steadily shrinking as the drivers are converted to using tasklets.

Once a BH has been marked, it is executed when *bh_action* (*kernel/softirq.c*) is invoked, which happens when tasklets are run. This happens whenever a process exits from a system call or when an interrupt handler exits. Tasklets are always executed as part of the timer interrupt, so a driver can usually expect that a bottom-half routine will be executed at most 10 ms after it has been scheduled.

Writing a BH Bottom Half

It's quite apparent from the list of available bottom halves in "The BH Mechanism" that a driver implementing a bottom half should attach its code to `IMMEDIATE_BH` by using the immediate queue.

When `IMMEDIATE_BH` is marked, the function in charge of the immediate bottom half just consumes the immediate queue. If your interrupt handler queues its BH handler to `tq_immediate` and marks the `IMMEDIATE_BH` bottom half, the queued task will be called at just the right time. Because in all kernels we are interested in you can queue the same task multiple times without trashing the task queue, you can queue your bottom half every time the top-half handler runs. We'll see this behavior in a while.

Drivers with exotic configurations—multiple bottom halves or other setups that can't easily be handled with a plain `tq_immediate`—can be satisfied by using a custom task queue. The interrupt handler queues the tasks in its own queue, and when it's ready to run them, a simple queue-consuming function is inserted into the immediate queue. See "Running Your Own Task Queues" in Chapter 6 for details.

Let's now look at the *short* BH implementation. When loaded with `bh=1`, the module installs an interrupt handler that uses a BH bottom half:

```
void short_bh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /* cast to stop 'volatile' warning */
    do_gettimeofday((struct timeval *) tv_head);
    short_incr_tv(&tv_head);

    /* Queue the bh. Don't care about multiple enqueueing */
    queue_task(&short_task, &tq_immediate);
    mark_bh(IMMEDIATE_BH);

    short_bh_count++; /* record that an interrupt arrived */
}
```

As expected, this code calls *queue_task* without checking whether the task is already enqueued.

The BH, then, performs the rest of the work. This BH is, in fact, the same *short_do_tasklet* that was shown previuosly.

Here's an example of what you see when loading *short* by specifying `bh=1`:

```
morgana% echo 1122334455 > /dev/shortint ; cat /dev/shortint
bh after      5
50588804.876653
50588804.876693
50588804.876720
50588804.876747
50588804.876774
```

The actual timings that you will see will vary, of course, depending on your particular system.

Interrupt Sharing

The notion of an IRQ conflict is almost synonymous with the PC architecture. In general, IRQ lines on the PC have not been able to serve more than one device, and there have never been enough of them. As a result, frustrated users have often spent much time with their computer case open, trying to find a way to make all of their hardware play well together.

But, in fact, there is nothing in the design of the hardware itself that says that interrupt lines cannot be shared. The problems are on the software side. With the arrival of the PCI bus, the writers of system software have had to work a little harder, since all PCI interrupts can explicitly be shared. So Linux supports shared interrupts—and on all buses where it makes any sense, not just the PCI. Thus, suitably aware drivers for ISA devices can also share an IRQ line.

The question of interrupt sharing under the ISA bus brings in the issue of level-triggered versus edge-triggered interrupt lines. Although the former kind of interrupt reporting is safe with regard to sharing, it may lead to software lockup if not handled correctly. Edge-triggered interrupts, on the other hand, are not safe with regard to sharing; ISA is edge triggered, because this signaling is easier to implement at hardware level and therefore was the common choice in the 1980s. This issue is unrelated to electrical signal levels; in order to support sharing, the line must be able to be driven active by multiple sources whether it is level triggered or edge triggered.

With a level-triggered interrupt line, the peripheral device asserts the IRQ signal until software clears the pending interrupt (usually by writing to a device register); therefore, if several devices pull the line active, the CPU will signal an interrupt as soon as the IRQ is enabled until all drivers have serviced their devices. This behavior is safe with regard to sharing but may lead to lockup if a driver fails to clear its interrupt source.

When using edge-triggered interrupts, on the other hand, interrupts may be lost: if one device pulls the line active for too long a time, when another device pulls the line active no edge will be generated, and the processor will ignore the second request. A shared handler may just not see the interrupt, and if its hardware doesn't deassert the IRQ line no other interrupt will be notified for either shared device.

For this reason, even if interrupt sharing is supported under ISA, it may not function properly; while some devices pull the IRQ line active for a single clock cycle, other devices are not so well behaved and may cause great pains to the driver writer who tries to share the IRQ. We won't go any deeper into this issue; for the rest of this section we assume that either the host bus supports sharing or that you know what you are doing.

To develop a driver that can manage a shared interrupt line, some details need to be considered. As discussed later, some of the features described in this chapter are not available for devices using interrupt sharing. Whenever possible, it's better to support sharing because it presents fewer problems for the final user. In some cases (e.g., when working with the PCI bus), interrupt sharing is mandatory.

Installing a Shared Handler

Shared interrupts are installed through *request_irq* just like nonshared ones, but there are two differences:

- The `SA_SHIRQ` bit must be specified in the `flags` argument when requesting the interrupt.
- The `dev_id` argument *must* be unique. Any pointer into the module's address space will do, but `dev_id` definitely cannot be set to `NULL`.

The kernel keeps a list of shared handlers associated with the interrupt, like a driver's signature, and `dev_id` differentiates between them. If two drivers were to register `NULL` as their signature on the same interrupt, things might get mixed up at unload time, causing the kernel to oops when an interrupt arrived. For this reason, modern kernels will complain loudly if passed a `NULL` `dev_id` when registering shared interrupts.

When a shared interrupt is requested, *request_irq* succeeds if either the interrupt line is free or any handlers already registered for that line have also specified that the IRQ is to be shared. With 2.0 kernels, it was also necessary that all handlers for a shared interrupt were either fast or slow—the two modes could not be mixed.

Whenever two or more drivers are sharing an interrupt line and the hardware interrupts the processor on that line, the kernel invokes every handler registered for that interrupt, passing each its own `dev_id`. Therefore, a shared handler must be able to recognize its own interrupts, and should quickly exit when its own device has not interrupted.

If you need to probe for your device before requesting the IRQ line, the kernel can't help you. No probing function is available for shared handlers. The standard probing mechanism works if the line being used is free, but if the line is already held by another driver with sharing capabilities, the probe will fail, even if your driver would have worked perfectly.

The only available technique for probing shared lines, then, is the do-it-yourself way. The driver should request every possible IRQ line as a shared handler and then see where interrupts are reported. The difference between that and do-it-yourself probing is that the probing handler must check with the device to see that the interrupt actually occurred, because it could have been called in response to another device interrupting on a shared line.

Releasing the handler is performed in the normal way, using *release_irq*. Here the `dev_id` argument is used to select the correct handler to release from the list of shared handlers for the interrupt. That's why the `dev_id` pointer must be unique.

A driver using a shared handler needs to be careful about one more thing: it can't play with *enable_irq* or *disable_irq*. If it does, things might go haywire for other devices sharing the line. In general, the programmer must remember that his driver doesn't own the IRQ, and its behavior should be more "social" than is necessary if one owns the interrupt line.

Running the Handler

As suggested earlier, when the kernel receives an interrupt, all the registered handlers are invoked. A shared handler must be able to distinguish between interrupts that it needs to handle and interrupts generated by other devices.

Loading *short* with the option `shared=1` installs the following handler instead of the default:

```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;

    /* If it wasn't short, return immediately */
    value = inb(short_base);
    if (!(value & 0x80)) return;

    /* clear the interrupting bit */
    outb(value & 0x7F, short_base);
    /* the rest is unchanged */

    do_gettimeofday(&tv);
    written = sprintf((char *)short_head,"%08u.%06u\n",
                      (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* wake any reading process */
}
```

An explanation is due here. Since the parallel port has no "interrupt-pending" bit to check, the handler uses the ACK bit for this purpose. If the bit is high, the interrupt being reported is for *short*, and the handler clears the bit.

The handler resets the bit by zeroing the high bit of the parallel interface's data port—*short* assumes that pins 9 and 10 are connected together. If one of the other devices sharing the IRQ with *short* generates an interrupt, *short* sees that its own line is still inactive and does nothing.

A full-featured driver probably splits the work into top and bottom halves, of course, but that's easy to add and does not have any impact on the code that implements sharing. A real driver would also likely use the `dev_id` argument to determine which, of possibly many, devices might be interrupting.

Note that if you are using a printer (instead of the jumper wire) to test interrupt management with *short*, this shared handler won't work as advertised, because the printer protocol doesn't allow for sharing, and the driver can't know whether the interrupt was from the printer or not.

The /proc Interface

Installing shared handlers in the system doesn't affect */proc/stat*, which doesn't even know about handlers. However, */proc/interrupts* changes slightly.

All the handlers installed for the same interrupt number appear on the same line of */proc/interrupts*. The following output shows how shared interrupt handlers are displayed:

```
           CPU0       CPU1
  0:   22114216   22002860    IO-APIC-edge  timer
  1:     135401     136582    IO-APIC-edge  keyboard
  2:          0          0          XT-PIC  cascade
  5:    5162076    5160039   IO-APIC-level  eth0
  9:          0          0   IO-APIC-level  acpi, es1370
 10:     310450     312222   IO-APIC-level  aic7xxx
 12:     460372     471747    IO-APIC-edge  PS/2 Mouse
 13:          1          0          XT-PIC  fpu
 15:    1367555    1322398    IO-APIC-edge  ide1
NMI:   44117004   44117004
LOC:   44116987   44116986
ERR:          0
```

The shared interrupt line here is IRQ 9; the active handlers are listed on one line, separated by commas. Here the power management subsystem ("acpi") is sharing this IRQ with the sound card ("es1370"). The kernel is unable to distinguish interrupts from these two sources, and will invoke each interrupt handlers in the driver for each interrupt.

Interrupt-Driven I/O

Whenever a data transfer to or from the managed hardware might be delayed for any reason, the driver writer should implement buffering. Data buffers help to detach data transmission and reception from the *write* and *read* system calls, and overall system performance benefits.

A good buffering mechanism leads to *interrupt-driven I/O*, in which an input buffer is filled at interrupt time and is emptied by processes that read the device; an output buffer is filled by processes that write to the device and is emptied at interrupt time. An example of interrupt-driven output is the implementation of */dev/shortint*.

For interrupt-driven data transfer to happen successfully, the hardware should be able to generate interrupts with the following semantics:

- For input, the device interrupts the processor when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping, or DMA.
- For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

The timing relationships between a *read* or *write* and the actual arrival of data were introduced in "Blocking and Nonblocking Operations", in Chapter 5. But interrupt-driven I/O introduces the problem of synchronizing concurrent access to shared data items and all the issues related to race conditions. The next section covers this related topic in some depth.

Race Conditions

We have already seen race conditions come up a number of times in the previous chapters. Whereas race conditions can happen at any time on SMP systems, uniprocessor systems, to this point, have had to worry about them rather less.*

---

* Note, however, that the kernel developers are seriously considering making *all* kernel code preemptable at almost any time, making locking mandatory even on uniprocessor systems.

Interrupts, however, can bring with them a whole new set of race conditions, even on uniprocessor systems. Since an interrupt can happen at any time, it can cause the interrupt handler to be executed in the middle of an arbitrary piece of driver code. Thus, any device driver that is working with interrupts—and that is most of them—must be very concerned with race conditions. For this reason, we look more closely at race conditions and their prevention in this chapter.

Dealing with race conditions is one of the trickiest aspects of programming, because the related bugs are subtle and very difficult to reproduce, and it's hard to tell when there is a race condition between interrupt code and the driver methods. The programmer must take great care to avoid corruption of data or metadata.

Different techniques can be employed to prevent data corruption, and we will introduce the most common ones. We won't show complete code because the best code for each situation depends on the operating mode of the device being driven, and on the programmer's taste. All of the drivers in this book, however, protect themselves against race conditions, so examples can be found in the sample code.

The most common ways of protecting data from concurrent access are as follows:

- Using a circular buffer and avoiding shared variables
- Using spinlocks to enforce mutual exclusion
- Using lock variables that are atomically incremented and decremented

Note that semaphores are not listed here. Because locking a semaphore can put a process to sleep, semaphores may not be used in interrupt handlers.

Whatever approach you choose, you still need to decide what to do when accessing a variable that can be modified at interrupt time. In simple cases, such a variable can simply be declared as `volatile` to prevent the compiler from optimizing access to its value (for example, it prevents the compiler from holding the value in a register for the whole duration of a function). However, the compiler generates suboptimal code whenever `volatile` variables are involved, so you might choose to resort to some sort of locking instead. In more complicated situations, there is no choice but to use some sort of locking.

Using Circular Buffers

Using a circular buffer is an effective way of handling concurrent-access problems; the best way to deal with concurrent access is to perform no concurrent access whatsoever.

The circular buffer uses an algorithm called "producer and consumer": one player pushes data in and the other pulls data out. Concurrent access is avoided if there is exactly one producer and exactly one consumer. There are two examples of producer and consumer in *short*. In one case, the reading process is waiting to consume data that is produced at interrupt time; in the other, the bottom half consumes data produced by the top half.

Two pointers are used to address a circular buffer: `head` and `tail`. `head` is the point at which data is being written and is updated only by the producer of the data. Data is being read from `tail`, which is updated only by the consumer. As mentioned earlier, if data is written at interrupt time, you must be careful when accessing `head` multiple times. You should either declare it as `volatile` or use some sort of locking.

The circular buffer runs smoothly, except when it fills up. If that happens, things become hairy, and you can choose among different possible solutions. The *short* implementation just loses data; there's no check for overflow, and if `head` goes beyond `tail`, a whole buffer of data is lost. Some alternative implementations are to drop the last item; to overwrite the buffer tail, as *printk* does (see "How Messages Get Logged" in Chapter 4); to hold up the producer, as *scullpipe* does; or to allocate a temporary extra buffer to back up the main buffer. The best solution depends on the importance of your data and other situation-specific questions, so we won't cover it here.

Although the circular buffer appears to solve the problem of concurrent access, there is still the possibility of a race condition when the *read* function goes to sleep. This code shows where the problem appears in *short*:

```
while (short_head == short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
    }
```

When executing this statement, it is possible that new data will arrive *after* the `while` condition is evaluated as true and *before* the process goes to sleep. Information carried in by the interrupt won't be read by the process; the process goes to sleep even though `head != tail`, and it isn't awakened until the next data item arrives.

We didn't implement correct locking for *short* because the source of *short_read* is included in "A Sample Driver" in Chapter 8, and at that point this discussion was not worth introducing. Also, the data involved is not worth the effort.

Although the data that *short* collects is not vital, and the likelihood of getting an interrupt in the time lapse between two successive instructions is often negligible, sometimes you just can't take the risk of going to sleep when data is pending. This problem is general enough to deserve special treatment and is delayed to "Going to Sleep Without Races" later in this chapter, where we'll discuss it in detail.

It's interesting to note that only a producer-and-consumer situation can be addressed with a circular buffer. A programmer must often deal with more complex data structures to solve the concurrent-access problem. The producer/consumer situation is actually the simplest class of these problems; other structures, such as linked lists, simply don't lend themselves to a circular buffer implementation.

Using Spinlocks

We have seen spinlocks before, for example, in the *scull* driver. The discussion thus far has looked only at a few uses of spinlocks; in this section we cover them in rather more detail.

A spinlock, remember, works through a shared variable. A function may acquire the lock by setting the variable to a specific value. Any other function needing the lock will query it and, seeing that it is not available, will "spin" in a busy-wait loop until it is available. Spinlocks thus need to be used with care. A function that holds a spinlock for too long can waste much time because other CPUs are forced to wait.

Spinlocks are represented by the type `spinlock_t`, which, along with the various spinlock functions, is declared in `<asm/spinlock.h>`. Normally, a spinlock is declared and initialized to the unlocked state with a line like:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

If, instead, it is necessary to initialize a spinlock at runtime, use *spin_lock_init*:

```
spin_lock_init(&my_lock);
```

There are a number of functions (actually macros) that work with spinlocks:

`spin_lock(spinlock_t *lock);`
: Acquire the given lock, spinning if necessary until it is available. On return from *spin_lock*, the calling function owns the lock.

`spin_lock_irqsave(spinlock_t *lock, unsigned long flags);`
: This version also acquires the lock; in addition, it disables interrupts on the local processor and stores the current interrupt state in `flags`. Note that all of the spinlock primitives are defined as macros, and that the `flags` argument is passed directly, not as a pointer.

`spin_lock_irq(spinlock_t *lock);`
: This function acts like *spin_lock_irqsave*, except that it does not save the current interrupt state. This version is slightly more efficient than *spin_lock_irqsave*, but it should only be used in situations in which you know that interrupts will not have already been disabled.

`spin_lock_bh(spinlock_t *lock);`
Obtains the given lock and prevents the execution of bottom halves.

`spin_unlock(spinlock_t *lock);`
`spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);`
`spin_unlock_irq(spinlock_t *lock);`
`spin_unlock_bh(spinlock_t *lock);`
These functions are the counterparts of the various locking primitives described previously. *spin_unlock* unlocks the given lock and nothing else. *spin_unlock_irqrestore* possibly enables interrupts, depending on the `flags` value (which should have come from *spin_lock_irqsave*). *spin_unlock_irq* enables interrupts unconditionally, and *spin_unlock_bh* reenables bottom-half processing. In each case, your function should be in possession of the lock before calling one of the unlocking primitives, or serious disorder will result.

`spin_is_locked(spinlock_t *lock);`
`spin_trylock(spinlock_t *lock)`
`spin_unlock_wait(spinlock_t *lock);`
spin_is_locked queries the state of a spinlock without changing it. It returns nonzero if the lock is currently busy. To attempt to acquire a lock without waiting, use *spin_trylock*, which returns nonzero if the operation failed (the lock was busy). *spin_unlock_wait* waits until the lock becomes free, but does not take possession of it.

Many users of spinlocks stick to *spin_lock* and *spin_unlock*. If you are using spinlocks in interrupt handlers, however, you must use the IRQ-disabling versions (usually *spin_lock_irqsave* and *spin_unlock_irqsave*) in the noninterrupt code. To do otherwise is to invite a deadlock situation.

It is worth considering an example here. Assume that your driver is running in its *read* method, and it obtains a lock with *spin_lock*. While the *read* method is holding the lock, your device interrupts, and your interrupt handler is executed on the same processor. If it attempts to use the same lock, it will go into a busy-wait loop, since your *read* method already holds the lock. But, since the interrupt routine has preempted that method, the lock will never be released and the processor deadlocks, which is probably not what you wanted.

This problem can be avoided by using *spin_lock_irqsave* to disable interrupts on the local processor while the lock is held. When in doubt, use the *_irqsave* versions of the primitives and you will not need to worry about deadlocks. Remember, though, that the `flags` value from *spin_lock_irqsave* must not be passed to other functions.

Regular spinlocks work well for most situations encountered by device driver writers. In some cases, however, there is a particular pattern of access to critical data that is worth treating specially. If you have a situation in which numerous threads (processes, interrupt handlers, bottom-half routines) need to access critical data in a read-only mode, you may be worried about the overhead of using spinlocks. Numerous readers cannot interfere with each other; only a writer can create problems. In such situations, it is far more efficient to allow all readers to access the data simultaneously.

Linux has a different type of spinlock, called a *reader-writer spinlock* for this case. These locks have a type of `rwlock_t` and should be initialized to `RW_LOCK_UNLOCKED`. Any number of threads can hold the lock for reading at the same time. When a writer comes along, however, it waits until it can get exclusive access.

The functions for working with reader-writer locks are as follows:

`read_lock(rwlock_t *lock);`
`read_lock_irqsave(rwlock_t *lock, unsigned long flags);`
`read_lock_irq(rwlock_t *lock);`
`read_lock_bh(rwlock_t *lock);`
: function in the same way as regular spinlocks.

`read_unlock(rwlock_t *lock);`
`read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);`
`read_unlock_irq(rwlock_t *lock);`
`read_unlock_bh(rwlock_t *lock);`
: These are the various ways of releasing a read lock.

`write_lock(rwlock_t *lock);`
`write_lock_irqsave(rwlock_t *lock, unsigned long flags);`
`write_lock_irq(rwlock_t *lock);`
`write_lock_bh(rwlock_t *lock);`
: Acquire a lock as a writer.

`write_unlock(rwlock_t *lock);`
`write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);`
`write_unlock_irq(rwlock_t *lock);`
`write_unlock_bh(rwlock_t *lock);`
: Release a lock that was acquired as a writer.

If your interrupt handler uses read locks only, then all of your code may acquire read locks with *read_lock* and not disable interrupts. Any write locks must be acquired with *write_lock_irqsave*, however, to avoid deadlocks.

It is worth noting that in kernels built for uniprocessor systems, the spinlock functions expand to nothing. They thus have no overhead (other than possibly disabling interrupts) on those systems, where they are not needed.

Using Lock Variables

The kernel provides a set of functions that may be used to provide atomic (noninterruptible) access to variables. Use of these functions can occasionally eliminate the need for a more complicated locking scheme, when the operations to be performed are very simple. The atomic operations may also be used to provide a sort of "poor person's spinlock" by manually testing and looping. It is usually better, however, to use spinlocks directly, since they have been optimized for this purpose.

The Linux kernel exports two sets of functions to deal with locks: bit operations and access to the "atomic" data type.

Bit operations

It's quite common to have single-bit lock variables or to update device status flags at interrupt time—while a process may be accessing them. The kernel offers a set of functions that modify or test single bits atomically. Because the whole operation happens in a single step, no interrupt (or other processor) can interfere.

Atomic bit operations are very fast, since they perform the operation using a single machine instruction without disabling interrupts whenever the underlying platform can do that. The functions are architecture dependent and are declared in `<asm/bitops.h>`. They are guaranteed to be atomic even on SMP computers and are useful to keep coherence across processors.

Unfortunately, data typing in these functions is architecture dependent as well. The `nr` argument is mostly defined as `int` but is `unsigned long` for a few architectures. Here is the list of bit operations as they appear in 2.1.37 and later:

`void set_bit(nr, void *addr);`
: This function sets bit number `nr` in the data item pointed to by `addr`. The function acts on an `unsigned long`, even though `addr` is a pointer to `void`.

`void clear_bit(nr, void *addr);`
: The function clears the specified bit in the `unsigned long` datum that lives at `addr`. Its semantics are otherwise the same as *set_bit*.

`void change_bit(nr, void *addr);`
: This function toggles the bit.

`test_bit(nr, void *addr);`
: This function is the only bit operation that doesn't need to be atomic; it simply returns the current value of the bit.

`int test_and_set_bit(nr, void *addr);`
`int test_and_clear_bit(nr, void *addr);`
`int test_and_change_bit(nr, void *addr);`

These functions behave atomically like those listed previously, except that they also return the previous value of the bit.

When these functions are used to access and modify a shared flag, you don't have to do anything except call them. Using bit operations to manage a lock variable that controls access to a shared variable, on the other hand, is more complicated and deserves an example. Most modern code will not use bit operations in this way, but code like the following still exists in the kernel.

A code segment that needs to access a shared data item tries to atomically acquire a lock using either *test_and_set_bit* or *test_and_clear_bit*. The usual implementation is shown here; it assumes that the lock lives at bit `nr` of address `addr`. It also assumes that the bit is either 0 when the lock is free or nonzero when the lock is busy.

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while();

/* do your work */

/* release lock, and check... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong(); /* already released: error */
```

If you read through the kernel source, you will find code that works like this example. As mentioned before, however, it is better to use spinlocks in new code, unless you need to perform useful work while waiting for the lock to be released (e.g., in the `wait_for_a_while()` instruction of this listing).

Atomic integer operations

Kernel programmers often need to share an integer variable between an interrupt handler and other functions. A separate set of functions has been provided to facilitate this sort of sharing; they are defined in `<asm/atomic.h>`.

The facility offered by *atomic.h* is much stronger than the bit operations just described. *atomic.h* defines a new data type, `atomic_t`, which can be accessed only through atomic operations. An `atomic_t` holds an `int` value on all supported architectures. Because of the way this type works on some processors, however, the full integer range may not be available; thus, you should not count on an `atomic_t` holding more than 24 bits. The following operations are defined for the type and are guaranteed to be atomic with respect to all processors of an SMP computer. The operations are very fast because they compile to a single machine instruction whenever possible.

`void atomic_set(atomic_t *v, int i);`
Set the atomic variable `v` to the integer value `i`.

`int atomic_read(atomic_t *v);`
Return the current value of `v`.

`void atomic_add(int i, atomic_t *v);`
Add `i` to the atomic variable pointed to by `v`. The return value is `void`, because most of the time there's no need to know the new value. This function is used by the networking code to update statistics about memory usage in sockets.

`void atomic_sub(int i, atomic_t *v);`
Subtract `i` from `*v`.

`void atomic_inc(atomic_t *v);`
`void atomic_dec(atomic_t *v);`
Increment or decrement an atomic variable.

`int atomic_inc_and_test(atomic_t *v);`
`int atomic_dec_and_test(atomic_t *v);`
`int atomic_add_and_test(int i, atomic_t *v);`
`int atomic_sub_and_test(int i, atomic_t *v);`
These functions behave like their counterparts listed earlier, but they also return the previous value of the atomic data type.

As stated earlier, `atomic_t` data items must be accessed only through these functions. If you pass an atomic item to a function that expects an integer argument, you'll get a compiler error.

Going to Sleep Without Races

The one race condition that has been omitted so far in this discussion is the problem of going to sleep. Generally stated, things can happen in the time between when your driver decides to sleep and when the *sleep_on* call is actually performed. Occasionally, the condition you are sleeping for may come about before you actually go to sleep, leading to a longer sleep than expected. It is a problem far more general than interrupt-driven I/O, and an efficient solution requires a little knowledge of the internals of *sleep_on.*

As an example, consider again the following code from the *short* driver:

```
while (short_head == short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

In this case, the value of `short_head` could change between the test in the `while` statement and the call to *interruptible_sleep_on*. In that case, the driver will sleep even though new data is available; this condition leads to delays in the best case, and a lockup of the device in the worst.

The way to solve this problem is to go halfway to sleep before performing the test. The idea is that the process can add itself to the wait queue, declare itself to be sleeping, and *then* perform its tests. This is the typical implementation:

```
wait_queue_t wait;
init_waitqueue_entry(&wait, current);

add_wait_queue(&short_queue, &wait);
while (1) {
    set_current_state(TASK_INTERRUPTIBLE);
    if (short_head != short_tail) /* whatever test your driver needs */
    break;
    schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(&short_queue, &wait);
```

This code is somewhat like an unrolling of the internals of *sleep_on*; we'll step through it here.

The code starts by declaring a `wait_queue_t` variable, initializing it, and adding it to the driver's wait queue (which, as you may remember, is of type `wait_queue_head_t`). Once these steps have been performed, a call to *wake_up* on `short_queue` will wake this process.

The process is not yet asleep, however. It gets closer to that state with the call to *set_current_state*, which sets the process's state to `TASK_INTERRUPTIBLE`. The rest of the system now thinks that the process is asleep, and the scheduler will not try to run it. This is an important step in the "going to sleep" process, but things still are not done.

What happens now is that the code tests for the condition for which it is waiting, namely, that there is data in the buffer. If no data is present, a call to *schedule* is made, causing some other process to run and truly putting the current process to sleep. Once the process is woken up, it will test for the condition again, and possibly exit from the loop.

Beyond the loop, there is just a bit of cleaning up to do. The current state is set to `TASK_RUNNING` to reflect the fact that we are no longer asleep; this is necessary because if we exited the loop without ever sleeping, we may still be in `TASK_INTERRUPTIBLE`. Then *remove_wait_queue* is used to take the process off the wait queue.

So why is this code free of race conditions? When new data comes in, the interrupt handler will call *wake_up* on `short_queue`, which has the effect of setting the state of every sleeping process on the queue to `TASK_RUNNING`. If the *wake_up* call happens after the buffer has been tested, the state of the task will be changed and *schedule* will cause the current process to continue running—after a short delay, if not immediately.

This sort of "test while half asleep" pattern is so common in the kernel source that a pair of macros was added during 2.1 development to make life easier:

`wait_event(wq, condition);`
`wait_event_interruptible(wq, condition);`

Both of these macros implement the code just discussed, testing the `condition` (which, since this is a macro, is evaluated at each iteration of the loop) in the middle of the "going to sleep" process.

Backward Compatibility

As we stated at the beginning of this chapter, interrupt handling in Linux presents relatively few compatibility problems with older kernels. There are a few, however, which we discuss here. Most of the changes occurred between versions 2.0 and 2.2 of the kernel; interrupt handling has been remarkably stable since then.

Differences in the 2.2 Kernel

The biggest change since the 2.2 series has been the addition of tasklets in kernel 2.3.43. Prior to this change, the BH bottom-half mechanism was the only way for interrupt handlers to schedule deferred work.

The *set_current_state* function did not exist in Linux 2.2 (but *sysdep.h* implements it). To manipulate the current process state, it was necessary to manipulate the task structure directly. For example:

```
current->state = TASK_INTERRUPTIBLE;
```

Further Differences in the 2.0 Kernel

In Linux 2.0, there were many more differences between fast and slow handlers. Slow handlers were slower even before they began to execute, because of extra setup costs in the kernel. Fast handlers saved time not only by keeping interrupts disabled, but also by not checking for bottom halves before returning from the interrupt. Thus, the delay before the execution of a bottom half marked in an interrupt handler could be longer in the 2.0 kernel. Finally, when an IRQ line was being shared in the 2.0 kernel, all of the registered handlers had to be either fast or slow; the two modes could not be mixed.

Most of the SMP issues did not exist in 2.0, of course. Interrupt handlers could only execute on one CPU at a time, so there was no distinction between disabling interrupts locally or globally.

The *disable_irq_nosync* function did not exist in 2.0; in addition, calls to *disable_irq* and *enable_irq* did not nest.

The atomic operations were different in 2.0. The functions *test_and_set_bit*, *test_and_clear_bit*, and *test_and_change_bit* did not exist; instead, *set_bit*, *clear_bit*, and *change_bit* returned a value and functioned like the modern *test_and_* versions. For the integer operations, `atomic_t` was just a `typedef` for `int`, and variables of type `atomic_t` could be manipulated like `int`s. The *atomic_set* and *atomic_read* functions did not exist.

The *wait_event* and *wait_event_interruptible* macros did not exist in Linux 2.0.

Quick Reference

These symbols related to interrupt management were introduced in this chapter.

```
#include <linux/sched.h>
int request_irq(unsigned int irq, void (*handler)(),
      unsigned long flags, const char *dev_name, void
      *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

These calls are used to register and unregister an interrupt handler.

```
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM
```

Flags for *request_irq*. `SA_INTERRUPT` requests installation of a fast handler (as opposed to a slow one). `SA_SHIRQ` installs a shared handler, and the third flag asserts that interrupt timestamps can be used to generate system entropy.

```
/proc/interrupts
/proc/stat
```

These filesystem nodes are used to report information about hardware interrupts and installed handlers.

```
unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
```

These functions are used by the driver when it has to probe to determine what interrupt line is being used by a device. The result of *probe_irq_on* must be passed back to *probe_irq_off* after the interrupt has been generated. The return value of *probe_irq_off* is the detected interrupt number.

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

A driver can enable and disable interrupt reporting. If the hardware tries to generate an interrupt while interrupts are disabled, the interrupt is lost forever. A driver using a shared handler must not use these functions.

```
DECLARE_TASKLET(name, function, arg);
tasklet_schedule(struct tasklet_struct *);
```

Utilities for dealing with tasklets. *DECLARE_TASKLET* declares a tasklet with the given `name`; when run, the given `function` will be called with `arg`. Use *tasklet_schedule* to schedule a tasklet for execution.

```
#include <linux/interrupt.h>
void mark_bh(int nr);
```

This function marks a bottom half for execution.

```
#include <linux/spinlock.h>
spinlock_t my_lock = SPINLOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
spin_lock(spinlock_t *lock);
spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
spin_lock_irq(spinlock_t *lock);
spin_lock_bh(spinlock_t *lock);
spin_unlock(spinlock_t *lock);
spin_unlock_irqrestore(spinlock_t *lock, unsigned long
      flags);
spin_unlock_irq(spinlock_t *lock);
spin_unlock_bh(spinlock_t *lock);
spin_is_locked(spinlock_t *lock);
spin_trylock(spinlock_t *lock)
spin_unlock_wait(spinlock_t *lock);
```

Various utilities for using spinlocks.

```
rwlock_t my_lock = RW_LOCK_UNLOCKED;
read_lock(rwlock_t *lock);
read_lock_irqsave(rwlock_t *lock, unsigned long flags);
read_lock_irq(rwlock_t *lock);
read_lock_bh(rwlock_t *lock);
read_unlock(rwlock_t *lock);
read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
read_unlock_irq(rwlock_t *lock);
read_unlock_bh(rwlock_t *lock);
```

`write_lock(rwlock_t *lock);`
`write_lock_irqsave(rwlock_t *lock, unsigned long flags);`
`write_lock_irq(rwlock_t *lock);`
`write_lock_bh(rwlock_t *lock);`
`write_unlock(rwlock_t *lock);`
`write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);`
`write_unlock_irq(rwlock_t *lock);`
`write_unlock_bh(rwlock_t *lock);`

The variations on locking and unlocking for reader-writer spinlocks.

`#include <asm/bitops.h>`
`void set_bit(nr, void *addr);`
`void clear_bit(nr, void *addr);`
`void change_bit(nr, void *addr);`
`test_bit(nr, void *addr);`
`int test_and_set_bit(nr, void *addr);`
`int test_and_clear_bit(nr, void *addr);`
`int test_and_change_bit(nr, void *addr);`

These functions atomically access bit values; they can be used for flags or lock variables. Using these functions prevents any race condition related to concurrent access to the bit.

`#include <asm/atomic.h>`
`void atomic_add(atomic_t i, atomic_t *v);`
`void atomic_sub(atomic_t i, atomic_t *v);`
`void atomic_inc(atomic_t *v);`
`void atomic_dec(atomic_t *v);`
`int atomic_dec_and_test(atomic_t *v);`

These functions atomically access integer variables. To achieve a clean compile, the `atomic_t` variables must be accessed only through these functions.

`#include <linux/sched.h>`
`TASK_RUNNING`
`TASK_INTERRUPTIBLE`
`TASK_UNINTERRUPTIBLE`

The most commonly used values for the state of the current task. They are used as hints for *schedule*.

`set_current_state(int state);`

Sets the current task state to the given value.

`void add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)`
`void remove_wait_queue(struct wait_queue ** p, struct wait_queue * wait)`
`void __add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)`
`void __remove_wait_queue(struct wait_queue ** p, struct wait_queue * wait)`

The lowest-level functions that use wait queues. The leading underscores indicate a lower-level functionality. In this case, interrupt reporting must already be disabled in the processor.

`wait_event(wait_queue_head_t queue, condition);`
`wait_event_interruptible(wait_queue_head_t queue, condition);`

These macros wait on the given queue until the given condition evaluates true.